Contents

1 Forelesning 1

1.1 Hva er en Algoritme?

En algoritme er en plan for å løse et problem, ikke helt det samme som en implementasjon. Den tar en input og fir en output (metode i *Java*).

Dagens Problem: Gitt en liste med brukernavn, sjekk at ingen brukernavn er like.

1.2 Forelesninger

Forelesningene blir tatt opp og gjort tilgjengelig på Mitt UiB, og kode blir lagt ut på git.

1.3 Pensum

Algorithms by Robert Sedgewick 4$^{\text{th}}$ edition. Kapittel 1-4 er pensum.

1. Fundamentale Algoritmer og datastrukturer
2. Sortering
3. Søking
4. Grafer

1.4 Beregne summen fra 1 til n

Følgende formel brukes for å regne summen fra 1 til n.

$$1 + 2 + 3 + \cdots + n = \sum_{i}^{n} \frac{n \cdot (n+1)}{2}$$

Kjøretiden til implementasjonen kan estimeres til

$$\frac{n \cdot (n+1)}{2} \cdot 20 \approx 10n^2$$

$$\begin{aligned} 10 \cdot n^2 &= 10^9 \\ &\approx 10000 \end{aligned}$$

Kjøretid på et tilsvarende program som bruker `Collections.sort()` vil ha en kjøretid på ca.

$$10 \cdot n \log n + 10 \cdot n = 10^9$$

Når n blir stor, er alltid $n \log n$ raskere enn n^2.

1.5 Hvordan forenkle beregninger?

Det er tidkrevende å kjøre programmet og ta tiden. Det er vanskelig å vite nøyaktig hvor mange operasjoner det er, derfor velger vi å beholde største ledd, og vi ser bort ifra konstanter. Som eksempel kan følgende formel forenkles.

$$\frac{n \cdot (n+1)}{2} \cdot 20 =\sim 10n^2 = O\left(n^2\right)$$

2 Forelesning 2

2.1 Hva er en datastruktur

En algoritme tar input, gjør beregninger, og gir output. Datastrukturer lagrer data og har flere forskjellige oppgaver som kan utføres på disse dataene.

Example. GPS Navigasjonsenhet

- Har veinettverk
- Kan be om korteste vei fra start til mål

Er dette en algoritme eller en datastruktur? ◇

`Collection <E> interface` er en samling med objekter av typen `E`. Viktige metoder er

- `size()`
- `contains(Object O)`
- `add (E e)`
- `Remove(Object obj)`
- `toArray()`

`List` er en utvidelse av `Collection` med litt flere metoder. Elementene i en liste har en indeks, og noen viktige metoder i `List` er

- `indexOf(Object obj)`
- `get(int index)`
- `set(int index, E e)`

2.2 Array List

I `ArrayList` brukes en array av typen `Object[]`. Bare en del av arrayen har data. Når dette arrayet blir fullt må vi lage et større array. Dette betyr at det er lett å få `IndexOutOfBoundsException`. For eksempel kan man lage et nytt array av dobbel størrelse, hvor alle elementer fra forrige array kopieres over til det nye arrayet.

2.3 Linked List

En Linked List er en liste hvor hvert element i listen lagres i et eget node object. Hver node vet kun neste og forrige node. Listen vet kun første og siste node. For å finne node i, så starter vi på første node og "hopper" i ganger.

En linked list som kun vet neste element kalles en *Single Linked List*, og en linked list som vet både neste og forrige element kalles en *Double Linked List*.

2.4 Hvordan finne kjøretid på metoder?

Man må vite hvordan `ArrayList` og `LinkedList` er implementert. Ved å forstå hva `ArrayList` og `LinkedList` gjør, er det lett å forstå hva kjøretiden er. Ukesoppgaven er og implementere enkle versjoner av disse listene.

2.5 Kø og Stabel

Kø	Stabel
FIFO	FILO / LIFO
Legger til sist	Legger til sist
Fjerner først	Fjerner sist

2.5.1 Metoder i Queue og Stack

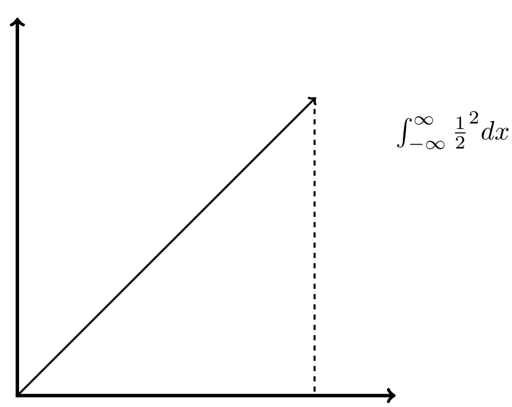
List	Queue	Stack
add(E e)	offer(E e)	push(E e)
remove(int i)	poll()	pop()
get(int i)	peek()	peek()

2.6 Set

Set er en Collection der hvert element kun kan være der en gang. Elementene har ikke en ebstemt ordning. Set har heller ikke `indexOf(element)` metoden.

	HashSet	TreeSet
`add()`	$O(1)^{\star}$	$O(\log n)$
`remove()`	$O(1)$	$O(\log n)$
`contains(obj)`	$O(1)$	$O(\log n)$

$$\int_{-\infty}^{\infty} \frac{1}{2}^2 dx = \nabla \times \vec{F} = \left| \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ \partial_x & \partial_y & \partial_z \end{bmatrix} \right| = \vec{i}\,(\partial_y - \partial_z) - \vec{j}\,(\partial_x - \partial_z) + \vec{k}\,(\partial_x - \partial_y)$$



3 Forelesning 3

3.1 Big O Kjøretid

Variablen n er vanligvis definert som størrelse på input, men den kan defineres til hva som helst. Når vi beregner kjøretid er $f(n)$ max antall opersjoner PCen må gjøre når et program kjøres på en input av størrelse n. Det vil si, velg den verste input av størrelse n. Teoretisk er vi interessert i

$$\sum_{n=0}^{\infty} f(n)$$

Mens praktisk er vi interessert i når $f(n)$ er 10^9 til 10^{12}. Vi har at

$$3 + 7 \leq 2 \cdot 7$$
$$124 + 98 \leq 2 \cdot 124$$

Dette gir oss følgende formel

$$a + b \leq 2 \cdot max(a, b)$$

Som resulterer i

$$O(1) + O(1) = O(1)$$
$$O(1) + O(n) = O(n)$$
$$O(n) + O\left(n^2\right) = O\left(n^2\right)$$

Big O beskriver et sett med funksjoner. $n^2 + 3n$ og $7n^2$ er i $O\left(n^2\right)$. Alle funksjoner som er mindre er også med, dvs at $3n$ og $7n \log(n)$ er i $O\left(n^2\right)$.

Definisjon 1. En funksjon $f(n)$ er i $O(g(n))$ dersom det finnes konstanter c og N slik at $f(n) < c \cdot g(n)$ for alle $n > N$, eller, skrevet rent matematisk

$$f(n) \in O(g(n)) \Rightarrow f(n) < c \cdot g(n) \, \forall \; n > N$$

3.2 Doubling Rate

Hva skjer med kjøretid når input blir dobbelt så stor?

- $O(1)$ - Ingenting, ikke avhengig av n.
- $O(n)$ - $c \cdot n \to c \cdot 2n$, dobbelt så lang tid.
- $O\left(n^2\right)$ - $c \cdot n^2 \to c \cdot (2n)^2$, fire ganger så stort.
- $O\left(n^3\right)$ - $c \cdot n^3 \to c\,(2n)^3$, 8 ganger så stort.

4 Forelesning 4

4.1 Rekursjon - Fibonacci

4.2 Binære trær

Et binært tres antall noder dobles for hvert nivå man går nedover treet.

4.3 Binærsøk

Problem: Sortert tabell med n element, leter etter et bestemt element. Hvor mange ganger må vi teste før vi er sikker på å finne det?

1. Forsøk - n element igjen: prøver midterste element
2. Forsøk - $\frac{n}{2}$ element igjen: prøver igjen midterste element

Osv, til det kun er ett element igjen.

4.4 Sortering - hvorfor?

4.4.1 Hva er sortering?

Sortering er en total ordning av en mengde A. F. eks følgende ordning

$$a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n$$

Ordningen kan være anti-symmetrisk.

$$-a \leq b \wedge a \geq b \Rightarrow a = b$$

Eller Transitiv

$$a \leq b \wedge b \leq c \rightarrow a \leq c$$

Eller Total

$$a \leq b \vee b \leq a$$

4.5 Sortering i Java

Man kan sortere med f. eks `Collections.sort(List <T> list)`, men hvilken type må `T` være? Man må implementere Comparable. Med Java generics blir det litt mer komplisert, men man kan implementere comparable på følgende måte.

```
T extends Comparable <? super T>
```

4.5.1 Comparable Interface

- `public int compareTo(T obj);`
  - Returnerer -1 hvis `this < obj`
  - Returnerer 0 hvis `this == obj`
  - Returnerer 1 hvis `this > obj`
- Beskriver naturlig ordning
  - Integer, Double, String, Date...

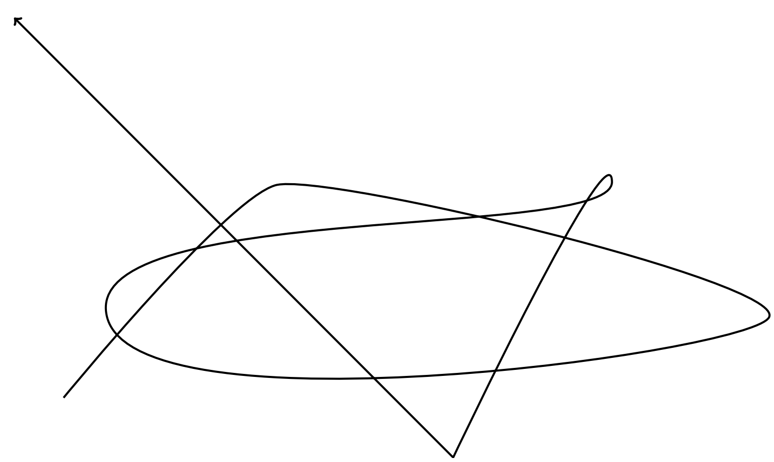
4.5.2 Sortere på forskjellige måter

Da må man bruke `Comparator interface`

```
int compare(T o1, T o2)
```

4.5.3 Mange sorteringsalgoritmer

- Selection sort
- Merge Sort
- Heap Sort
- Bogo Sort (The GOAT)
- Insertion Sort
- Shell Sort
- Quick Sort
- Bucket Sort
- Radix Sort
- Bubble Sort



5 Forelesning 5

5.1 Selection Sort

- Finn det minste elementet
- Flytt det først (index 0)

- Finn minste elementet blant de $n - 1$ siste.
- Flytt det på andre posisjon (index 1)

$\cdots$ til alle elementene er sortert.

Hva er kjøretiden til denne funksjonen?

```java
public <T extends Comparable<? super T>> void sort(List<T> list) {
    List<T> sorted = new ArrayList<>();

    while (!list.isEmpty()) { // n iterasjoner
        T smallest = Collections.min(list); // O (n)
        list.remove(smallest); // O (n)
        sorted.add(smallest); // O (n)
    }
    list.addAll(sorted);
}
```

Figure 1: Selection sort har kjøretid $O\left(n^2\right)$

5.2 "In-Place" Sortering

- Sortere uten å bruke ekstra minne
- Ikke lag ny liste, men bytt elementer i listen
- Dette forbedrer ikke big-O kjøretid, men kan redusere konstantene i kjøretiden
- Dette kalles optimering av kode

5.3 Insertion Sort

- Lag en ny liste
- Gå gjennom elementene

5.4 Quick Sort

- Finn et element kalt "pivot"
- Divide:
  - Første list $\leq$ pivot
  - Siste list $\geq$ pivot
  - Hva med de lik pivot?
- Sort hver liste rekursivt.
- Slå sammen disse listene

  - Siden vi delte inn i basert på pivot blir det
  - Første, pivot, siste
- Hva er kjøretiden til QuickSort
  - Det ligner litt på merge sort
  - Vi deler opp i to lister, men er de like store?
  - Hvor mange ganger må vi dele opp?
- I verste tilfelle blir pivot det største eller minste elementet?
- I beste tilfelle blir pivot det midterste elementet.

Quicksort har altså runtime $T \in \left(O\left(n\log\left(n\right)\right), O\left(n^2\right)\right)$

6 Forelesning 6

6.1 Divide & Conquer

- En nyttig strategi når man skal finne ffektive algoritmer.
- Divide
  - Del opp input og løs hver del for seg
- Conquer
  - Finn en måte å slå sammen de to delene til løsning for hele input

6.2 Mergesort

- Divide
  - Lag to lister som hver har halvparten av elementene
  - Sorter de to små listene
- Conquer
  - Gitt to små sorterte lister, slå disse sammen til en stor sortert liste.
- La oss implementere mergesort.

6.3 Nedre grense på kjøretid

- Vi kan se å sortering som å velge den rette ordningen blant alle mulige ordninger av en liste.
- Hvor mange måter er det å ordne n elementer? ($n!$)
- Tenk deg en liste av alle mulige ordninger
- Gjør en sammenligning av 2 elementer a og b.
- Blant alle ordninger i listen kan vi
  - Enten Krysse ut de der $a < b$
  - Eller krysse ut de der $b < a$
- Hvor mange ganger må vi dele på 2 før $n!$ blir 1?
- $\log\left(n!\right)$

6.4 Sorter n heltall

- Hvor for kan vi sortere n heltall der alle tallene er
  - Mellom 0 og 1000
  - Mellom 0 og n
  - Mellom 0 og n^2

6.5 Bucket Sort

- Ikke egentlig en sorteringsalgoritme
- Deler opp i grupper (buckets/bins)
  - For alle par a, b har vi: $a \leq b \leftrightarrow \text{bucket}(a) \leq \text{bucket}(b)$
- Trenger funksjon for bucket nummer
  - Eksempel: String, 26 buckets, a i første, b i andre.
- Radix sort ligner på bucket sort
  - Deler igjen opp hver bucket basert på 2. bokstav.

7 Forelesning 7

7.1 Problem vi skal løse idag

7.2 Priority Queue

- En datastruktur med følgende operasjoner
  - `add(T element)`
  - `T findMin()` (eller `findMax`)
  - `T removeMin()` (eller `removeMax`)

7.2.1 Linked list

- `add(T element)` - $O(1)$
- `T findMin()` - $O(n)$
- `T removeMin()` - $O(n)$

7.2.2 Sortert Liste

Data-invariant: Listen er sortert

- `add(T element)` - $O(n)$
- `T peekMin()` - $O(1)$
- `T removeMin()` - $O(1)$

7.3 Eksempel på bruk av priority queue

- Kø på legevakten
  - De som er mest syk kommer til først
- Operativsystem
  - Viktige prosesser får høy prioritet
- Graf algoritmer - mer i kaptitel 4
- Huffman data compression
  - De vanligste sekvensene får kort kode
  - De minst vanlige får lang kode

7.4 Binary Tree - Datainvariant

- 1 root
- Hver node har opp til 2 children
- Hvis a er child av b så sier vi at b er parent av a
- Hver node har 1 parent untatt root som har 0.
- De nederste nodene kalles *leaf*

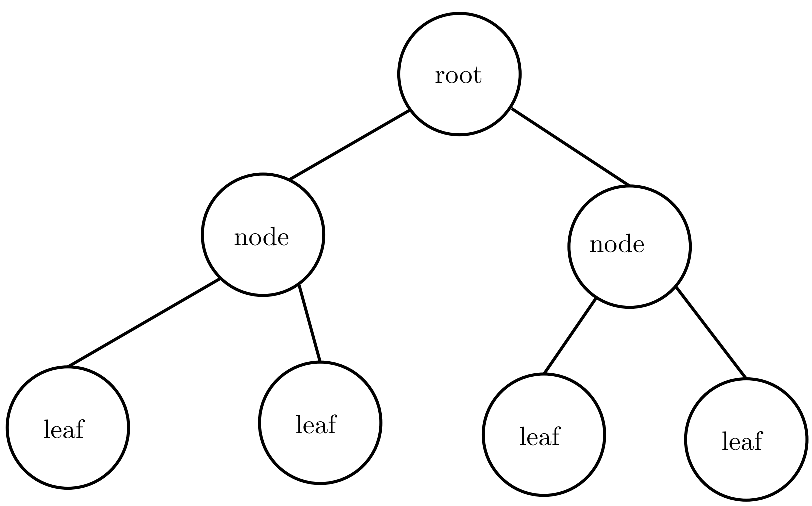


Figure 2: Binært tre

7.5 Complete binary tree

Et komplett binært tre består av halvparten som løv, og den andre halvparten av noder og røtter. Høyden til et komplett binært tre er $\log(n)$.

7.6 Heap - data invariant

- Er et binary tree med følgende krav:
  - Hver node er mindre enn alle sin children
  - Treet er balansert

7.6.1 Kjøretid

- `add(T element)` - $O(\log(n))$
- `T peekMin` - $O(1)$
- `T removeMin()` - $O(\log(n))$

7.7 Heap Sort

- Kan vi bruke Heap til å sortere?
- La oss implementere HeapSort.

7.8 Median Data Structure

Use sorted ArrayList

- `add()` - $O(n)$
- `findMedian()` - $O(1)$

Bruk to heap datastrukturer, en larger alle elementer midnre enn median, den andre lagrer alle elementer større enn median.

- `add()` - $O(\log n)$
- `findMedian()` - $O(1)$

7.9 Heap - Initialize

Gitt en liste, hvor for kan vi legge inn alle elementene i en Heap?

```
for (T elem : list)
    heap.add(elem)
```

Dette gir kjøretid $O(n \log(n))$

7.9.1 Kjøretid

- Nederste level: 0
- Nest nederste: $\frac{n}{2} \cdot 1$
- Nest nest nederste: $\frac{n}{4} \cdot 2$

Totalt:

$$\sum_{i=1}^{\log(n)} \frac{n \cdot i}{2^i} \in O(n)$$

8 Forelesning 8

8.1 Set datastruktur

- Set er nesten list, bare at i set er alle elementer unike.
- Viktige metoder
  - Add
  - Remove
  - Contains
- Set i Java:
  - TreeSet
  - HashSet

8.2 Binary search tree

- Binært søketre på norsk
- Den datastrukturen som er brukt i TreeSet
- Ligner veldig på Heap
- Kan kun brukes med Comparable
- Data invariant:
  - Verider til venstre er mindre enn roten
  - Verdier til høyre er større enn roten

8.2.1 Balansert

- Siden kjøretid er høyden av treet må vi prøve å holde høyden til $O(\log(n))$
- Balansering kan gjøres på mange måter.
- Beskrives i kapittel 3.3 i boken
- Hvis man finner en node der forskjellen mellom antall noder til venstre og antall noder til høyre er stor, da må man flytte fra høyre til venstre
- Det å balansere et tre kan ta litt mer en $O(\log(n))$ tid
- Som i ArrayList sin "grow" metode, trengs balansering ikke gjøres hver gang.
- Detaljene i 3.3 er ikke relevant ofr eksamen, men viktig å forstå at balansering er nødvendig for $\log(n)$ kjøretid.

8.3 HashSet

- Kan brukes på alle objekter i Java
- Krever metoden `hashCode()`
  - Mer om den senere
- Bruker mer minne, men er rask
- Ikke effektivt hvis du trenger å finne max/min

9 Forelesning 10

9.1 Graf

Grafer er en datastruktur som består av

- Noder (Vertices)
- Kanter (Edges) - par av noder

Binære trær er grafer.

9.2 Hva kan grafer representere?

Noder	**Kanter**
Byer	Veier
Person	Vennskap
Datamaskin	Nettverkskabel
Spill posisjon	Gyldig trekk
Student, emnet	Tar emnet
Robot, Job	Assigned
Lag	Kamp

9.2.1 Gradtall

Antall kanet som inneholder noden

9.2.2 Naboskap

Alle noder som kan nås med en kant

$$N(b) = \{a, c, f\} \wedge N[b] = \{a, b, c, f\}$$

9.2.3 Sykel

Noder og kanter som gjør at du kan gå fra en node og tilbake til seg selv mens du bare besøker hver node i sykelen akkurat en gang.

9.2.4 Tre

Et tre er en sammenhengende graf uten sykler.

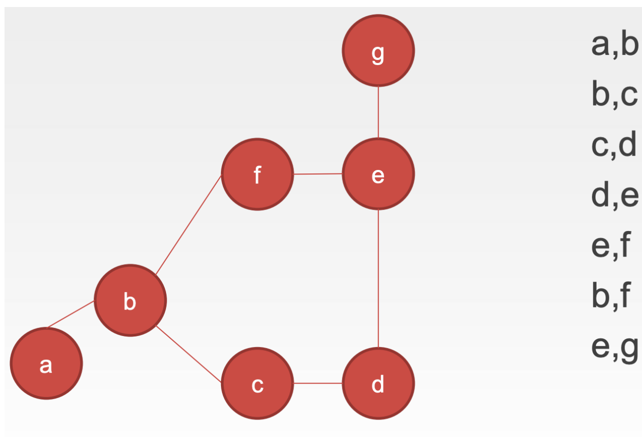
9.3 Graf datastruktur

- Viktige metoder i `Graph<V,E>`
  - `boolean adjacent (V a, V b)`
  - `Iterable<V> vertices()`
  - `Iterable<E> edges()`
  - `List<V> neighbours(V v)`
  - `add/remove metoder?`

9.3.1 Kjøretid i grafer

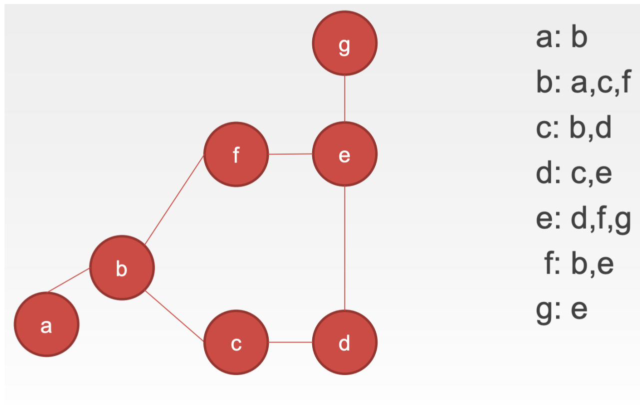
- Kjøretid i grafer avhenger av 2 parametere
  - N er antall noder
  - M er antall kanter
- Noen ganger er man mer presis
  - D = degree(v)
  - L = length of longest cycle
  - C = number of components

9.3.2 Graf datastruktur - Kantlise



Metode	**Kjøretid**
Ajacent	$O(m)$
Vertices	$O(n)$
Edges	$O(m)$
Neighbours	$O(m)$
addVertex	$O(1)$
addEdge	$O(1)$

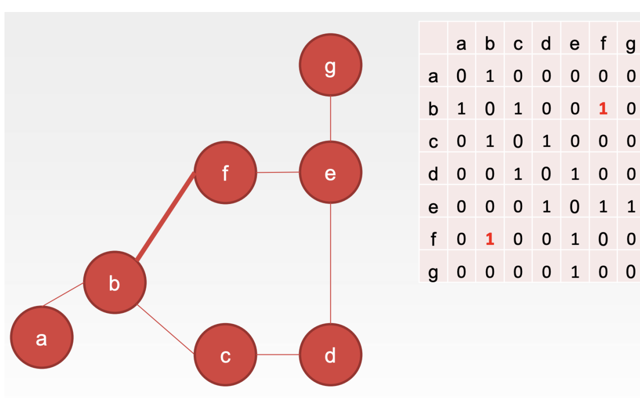
9.3.3 Graf datastruktur - Naboliste



- For hver node, hold liste av naboer
- Hold en liste av noder `List<V>`

Metode	**Kjøretid**
Adjacent	$O\,(\text{degree})$
Vertices	$O\,(n)$
Edges	$O\,(m)$
Neighbours	$O\,(\text{degree})$
addVertex	$O\,(1)$
addEdge	$O\,(1)$

9.3.4 Graf datastruktur - Nabomatrise



- 2 dimensjonell boolean tabell
- Bruker mye minne

Metode	Kjøretid
Adjacent	$O(1)$
Vertices	$O(n)$
Edges	$O(n^2)$
Neighbours	$O(n)$
addVertex	$O(n^2)$ or $O(n)$
addEdge	$O(1)$

9.4 Sammenhengende komponenter

Hvordan kan vi finne alle sammenhengende komponenter? Vi skal nå se på en algoritme som finner sammenhengende komponenter i en graf.

1. Søke i grafen med noe vi kaller bredde først søk.
2. Søke i grafen med noe vi kaller dybde først søk.
3. Bruke Union-Find (vil lære om dette snart)

10 Forelesning 11

10.1 Bredde først søk (BFS)

Hver gang man implementerer noe har man to valg, enten gjøre det iterativt, eller rekursivt. Ofte blir koden man skriver lettere dersom man bruker rekursjon.

DFS egner seg godt til en rekursiv implementasjon, og BFS egner seg ofte godt til en iterativ implementasjon.

10.1.1 La oss implementere BFS

- Jeg skal bruke grafklassen som dere har fått utdelt med semesteroppgave 2

10.2 Vektet graf

- Ofte er det naturlig å ha vekter på kantene
  - For eksempel noder er byer, kanter er veier
  - Vekt på kant er lengde på veien i km
- Vi kan ønske å finne stien med minst sum av vekter.
  - Korteste vei å kjøre fra en by til en annen.

10.3 Minimum spanning tree

- En sub-graf er en graf som kan lages fra en annen graf ved å fjerne kanter.
- Et tre er en sammenhengende graf uten sykler
- Et sub-tre er en subgraf som er et tre
- Et spanning sub-tre er et sub-tre der alle nodene er med (ingen noder er slettet)

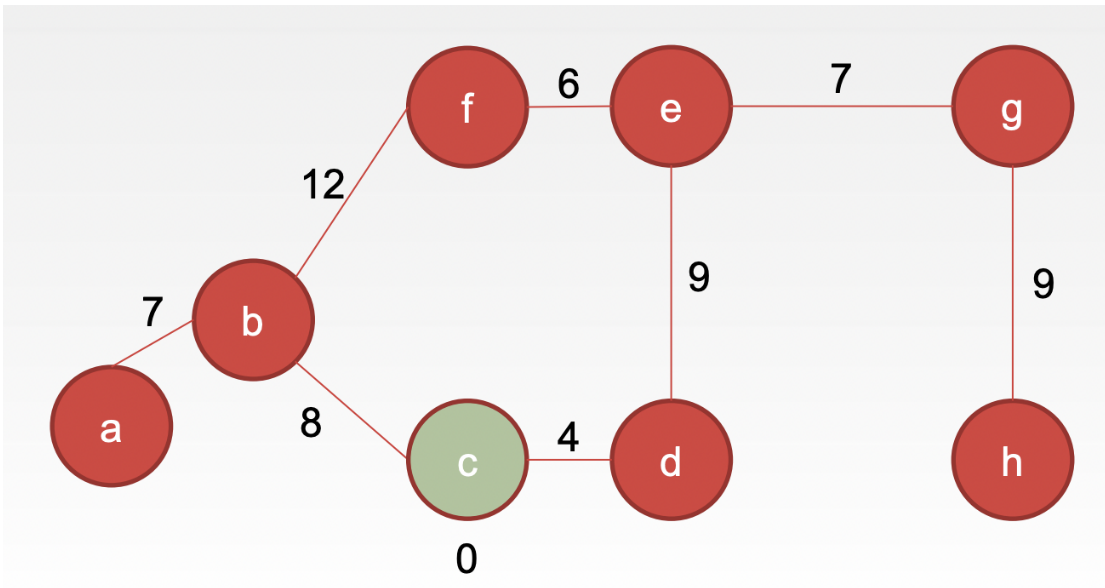
En av oppgavene i Semesteroppgave 2 er å implementere **Prims algoritme**.

11 Forelesning 12

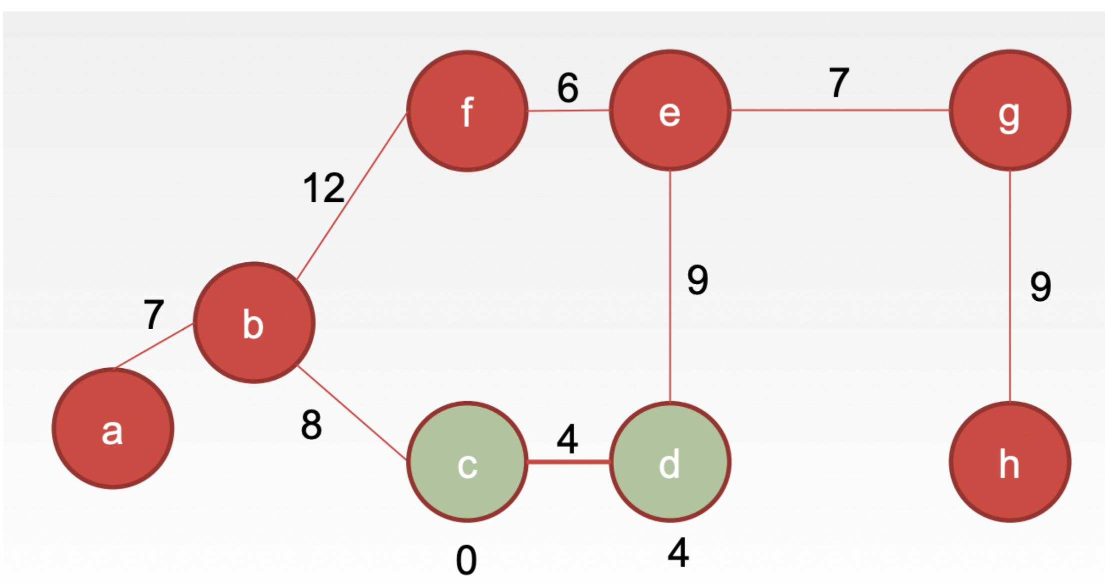
11.1 Finn korteste sti

Gitt en graf, finn koteste sti fra en node til en annen, f. eks node c til node h.

11.1.1 Dijkstras algorithm



found:c, d toSearch : $(c, d) : 8, (d, e) : 13$



... osv

11.2 Minimum Spanning Tree

Et minimum spanning tree av grafen G er en delmenge T av kantene i G slik at:

- Alle par av noder har en sti mellom seg som kun bruker kanter i T.
- T er den delmenged med minst sum av vekter over alle kanter i T.

Forrige uke fant vi MST med Prims algoritme.

11.3 Hvordan vite om sykel oppstår?

- I Prim's algorithme sjekket vi om begge endene var i found
  - Dette virket fordi vi startet fra en node
  - Og kanten vi så på hadde 1 ende i found
  - ALle nodene i found var i samme komponent
- I Kruskal's har vi mange sammenhengende komponenter
- Vi må sjekke om begge endepunktene er i samme komponent!

11.3.1 Hvilke metoder trenger vi

Union Find interface:

- `public E find (E elem);`
- `public void union (E elem1, E elem2);`
- `public int found();`
- `Iterable<E> group(E elem);`

12 Forelesning 13

12.1 Oblig 1 - Oppgave 3

Hvorfor er `ClosestStrategy` ikke optimal?

- Første observasjon er at det for noen input er stor `backLog` mens for andre er den liten.
- Gjør vi små jobber blir flere jobber ferdig fort
- Små jobber ligger nært roboter.

Hvor mange jobber er på `backLog`? Det kan være lurt å legge til metoder for å hente ut ekstra statistikk.

Maks størrelse på `backLog`

- Input 1: 129
- Input 2: 656
- Input 3: 0
- Input 4: 302

- Input 5: 2
- Input 6: 0

For de input med mye på `backLog`

- Hva skjer om vi velger minste jobb først?

13 Forelesning 14

13.1 Rettede grafer

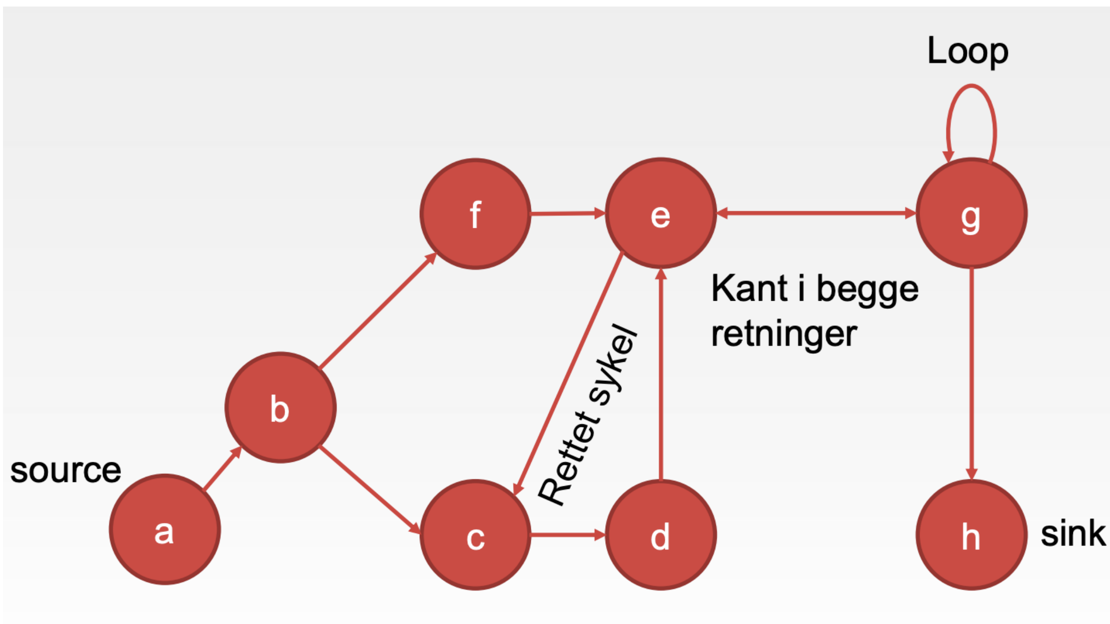


Figure 3: Eksempel på en rettet graf

Noder som kun har utgående kanter kalles `source`, og noder som kun har inngående noder kalles `sink`.

13.1.1 Nye metoder i rettede grafer

- `Iterable<V> outNeighbours (V v)`
- `Iterable<V> inNeighbours (V v)`
- `int outDegree`

13.1.2 BFS og DFS i rettet graf

- Virker akkurat som i urettet graf
- Der du før brukte `neighbours()`, må du nå bruke `outNeighbours()`
- DFS kan burkes til å finne en rettet sykel
  - Ser dere hvordan?
  - Jo, hvis vi kommer til en node som allerede ligger i `found`.

13.2 Strongly connected components

- Forkortes SCC
- Det finnes sti fra a til b og sti fra b til a, hvis og bare hvis a og b er i samme SCC.
- Det vil si at en strongly connected component er et sett med noder der det alltid finnes en sti du kan følge for å komme dit du vil.

13.3 Directed Acyclic Graph

- En graf uten noen rettede sykler kaller vi en DAG.
- Alle nodene i en DAG kan ordnes slik at alle kanter går fra venstre til høyre.
- Hvordan kan vi finne en slik ordning?

13.3.1 Topological sort (finn ordning)

```
int i = 0
while (g has a source)
    let v be a source
    order(i) = v
    i++
    remove v from g
```

13.3.2 Topological sort kjøretid

- Kan vi finne neste source fortere enn O(n)?
  - Ja, ha en liste over alle som har gradtall 0.
  - Hver gang gradtall oppdateres
    - Hvis ny verdi er 0, legg til i liste
  - Total $O(m)$
    - Oppdatering $O(m)$
    - $O(n)$ flyttinger til/fra liste